Alpha Seeding for Support Vector Machines

Dennis DeCoste
Machine Learning Systems Group
Jet Propulsion Laboratory / California Institute of Technology
4800 Oak Grove Drive; Pasadena, CA 91109
decoste@aig.jpl.nasa.gov, http://www-aig.jpl.nasa.gov/home/decoste/

Kiri Wagstaff
Department of Computer Science, Cornell University
4156 Upson Hall; Ithaca, NY 14853
wkiri@cs.cornell.edu, http://www.cs.cornell.edu/home/wkiri/

Abstract

A key practical obstacle in applying support vector machines to many large-scale data mining tasks is that SVM's generally scale quadratically (or worse) in the number of examples or support vectors. This complexity is further compounded when a specific SVM training is but one of many, such as in Leave-One-Out-Cross-Validation (LOOCV) for determining optimal SVM kernel parameters or as in wrapper-based feature selection. In this paper we explore new techniques for reducing the amortized cost of each such SVM training, by seeding successive SVM trainings with the results of previous similar trainings.

1 Introduction

Recent progress on speeding up the training time for support vector machines (e.g. [8],[5]) has made SVM's practical now for training sets that are fairly large. However, the time complexities of those approaches are still typically quadratic in the number of examples (N) in the training data set. This is especially problematic in a data mining context, due both to commonality of enormous data set sizes and to the frequent need for high-quality model selection over many candidate SVM's.

Given that the complexity of the best methods for training a single SVM tend to be quadratic in N, we seek methods which could reuse the results from training some SVM when training similar SVM's, in the hopes of amortizing that cost. In the best case, this might lead to amortized SVM training costs which are linear in N. For example, Leave-One-Out-Cross-Validation (LOOCV) estimates of generalization error for a data set of N examples involve N trainings, each involving $N-1$ training examples. If each SVM for each of the size $N-1$ data sets could be intelligently initialized from the result of the SVM trained on all N examples, only a small amount of additional work might be required for each. The overall cost might well remain quadratic in N (i.e. dominated by the cost of the SVM trained on the full data set) — and thus effectively have cost linear in N for each of the N SVM's trained for the different size $N-1$ data sets.

An underlying motivation of our work is to try to bring SVM's substantially closer to the fast linear complexity of LOOCV using k nearest-neighbors, (a factor in k-NN's popularity in practice), while retaining the advantages of SVM's (e.g. maximum margins). [1]

After reviewing the basic aspects of SVM classification, we will present a variety "alpha seeding" methods for reducing SVM training time. We will then present some empirical results which illustrate the potential promise of such alpha seeding and help us begin to understand the tradeoffs involved. Although we have not yet achieved linear amortized costs, our results appear promising towards that effort, as well as of practical use in their own right.

[1] For Euclidian distance, complexity logarithmic in N is often achieved for k-NN's, using indexing schemes such as k-d trees. However, for the general distance metrics employed within SVM kernel methods [9, 3] sub-linear performance for k-NN's is not as obviously achieved.

2 Support Vector Machines

Support vector machines [10, 11] represent a relatively new and promising approach to machine learning. Recent work has established SVM's as providing state-of-the-art performance on classification and regression tasks across a variety of real-world applications (e.g. see [10] and [4]).

In this paper, we focus on SVM's for binary classification [1]. In binary classification, each label is valued either "+1" or "-1", indicating either a positive or negative example, respectively.

Let X_A be an n_A by D matrix representing the training set and X_B be an n_B by D matrix representing the test set, where D is the dimensionality of the input space (i.e. D features) and n_A and n_B are the number of training and test examples, respectively. Let L_A be a vector of the n_A known labels for the training set and L_B be a vector of the n_B actual (often unknown) labels for the test set. Let y_B be a vector of the n_B label predictions of the automated classifier for the test set X_B. Furthermore, let cost(y_B,L_B) reflect task-specific relative costs of false positives versus false negatives.

The goal is to train a classifier from given examples X_A and labels L_A that minimizes the (expected) value for cost(y_B,L_B), for the test set X_B.

2.1 Basics of SVM Classification

The following constrained quadratic optimization (QP) problem is commonly used to train a SVM classifier:

$$
\begin{aligned}
&\textit{maximize:} \\
&\qquad \textstyle\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j L_i L_j K(x_i, x_j) \\
&\textit{subject to:} \\
&\qquad 0 \leq \alpha_i \leq C^+ \text{ if } L_i = +1, \\
&\qquad 0 \leq \alpha_i \leq C^- \text{ if } L_i = -1, \\
&\qquad \textstyle\sum_{i=1}^{N} \alpha_i L_i = 0,
\end{aligned}
$$

using notational simplifications: $N = n_A$, $L = L_A$, and x_i is i-th example (row) in X_A. This is consistent with recent approaches (e.g. [12]) for imbalanced sets of negative and positive examples.

The prediction of the SVM, for any example x (vector of size D), is given by:

$$f(x) = sign(\sum_{i=1}^{n} \alpha_i L_i K(x, x_i) + b,$$

where scalar b (bias) and vector α (of size n) contains the variables determined by the above QP optimization problem. For example, the test predictions y_B are $f(x)$, for each x in X_B.

Scalars C^+ and C^- are two parameters fixed before performing QP optimizations. The ratio C^+/C^- represents task-specific knowledge of how much more costly false negatives (e.g. missed events) are to false positives (e.g. false alarms). Their specific values represents the costs of overfitting noise in the training data. In our work, we typically determine one (say C^-) empirically, using LOOCV over various candidate values. For the scope of this paper, we will focus on the special case of $C = C^- = C^+$ and simply refer to a single parameter C.

$K(x_i, x_j)$ represents a *kernel* which implicitly projects two given examples from D dimensional input space into some (possibly infinite, typically nonlinear) feature space. The simplest is the *linear* kernel, implemented as a simple dot product:

$$K(u, v) \equiv u \otimes v \equiv \sum_{i=1}^{d} u_i \cdot v_i.$$

The *polynomial* kernel is defined by a non-linearly squashed dot-product of the following form:

$$K(u, v) \equiv (u \otimes v + r)^d,$$

with polynomial degree parameter d. Varying the continuous offset parameter r changes the relative weighting of the (implicit) terms in the nonlinear polynomial feature space.

One of the most popular kernels is the *radial basis function* (RBF) nonlinear kernel:

$$K(u,v) \equiv e^{\frac{-||u-v||^2}{2\sigma^2}},$$

with a variance parameter σ, which is also based on different non-linear squashing of the dot-product between two examples [2].

Reasonable settings for kernel parameters such as d, r and σ above can often be determined using either theoretical estimates of the generalization error (e.g. via Vapnik's bounds based on VC-dimension) or empirical estimation methods such as LOOCV.

Support vectors are those training example vectors for which $\alpha_i > 0$. As can be seen from the above summation used to generate predictions, a zero α_i means that the i-th training example does not contribute to the prediction. In SVM applications often only 10% or less of the training examples become supports. Such sparsity is a key property of SVM's that helps them avoid overfitting noise. A general rule of thumb is that the expected test error of the SVM is proportional to the ratio of the number of support vectors to the number of all training examples.

3 Types of Alpha Seeding

We use the term *alpha seeding* to refer to any method which provides initial estimates of the alpha (α) values for the SVM's QP optimization problem and starts the QP problem using them, instead of using the default of all zero alphas that existing SVM methods use. We will restrict ourselves to methods which start each SVM training with *feasible* alphas (i.e. which satisfy the bounds and the single equality constraint), although it is conceivable that infeasible seeds may be useful in some contexts for some specific SVM training algorithms.

To motivate our work and establish a framework, below we discuss a variety of ways in which the alpha seeding can be used to improve various aspects of SVM training. In Section 4, we will empirically explore some of these in more detail.

The methods we have identified fall into two broad classes, which we refer to as *sequential* and *branching*. Most methods we have identified are sequential and incremental in nature — they involve estimating a series of alpha sets, with the seeds for the next SVM training being based on the results of the previous similar SVM training. In some cases, in particular estimating LOOCV errors, the flow of alpha seeds follows a branching tree structure, rather than a chain.

A fundamental issue is in how the alphas from a previous training should be adapted into appropriate seeds for the next training. As we shall explore, there are typically much more effective approaches than simply passing the alphas unchanged between trainings.

All the tasks for which we introduce alpha seeding methods can be solved without seeding (i.e. just start each with zero alphas). Thus, alpha seeding offers no new theoretical advance, as, say, a new type of SVM kernel might. Instead, the goal of alpha seeding is drastically faster convergence to the final alpha values for the SVM problem(s) of interest. However, it is important to keep in mind that resource allocation is almost always a concern in practice. For example, if one can speed the SVM training for one kernel or C value by a factor of 10, one may be able to search for the optimal of ten different types of kernels (or C values) in the same fixed available overall training time.

It is also useful to keep in mind that all of these approaches to alpha seeding can amortize the cost of kernel computations across the entire set of of SVM trainings. Dot-product caches are common even for single SVM trainings, as in most practical SVM trainers (e.g. [5]). Our alpha seeding techniques exploit dot-product caches even further, with the later trainings often requiring no additional kernel computations. When input dimensionality D is large, these savings can be very substantial (typically more than 200% versus no cache).

The key issue determining whether a given alpha seeding method is effective for a given task is, of course, whether the sum of the training costs over the sequence of successively seeded SVM's is lower than the cost of directly training the non-seeded SVM of interest. We will explore that issue in Section 4, after first discussing the various methods.

[2] Where $||u-v||^2 \equiv (u \otimes u - 2u \otimes v + v \otimes v)$.

3.1 Computing Actual LOOCV Error

One of simplest and yet effective alpha seeding methods is for efficient LOOCV estimation of generalization error. LOOCV requires N SVM trainings, where the i-th SVM is tested on the i-th example and is trained only on the $N-1$ other examples. Unlike other methods below, each such case is for fixed parameters (e.g. for given C, RBF kernel σ, etc.). Doing multiple LOOCV's, for various parameter values, provides a popular empirical-based means of model selection.

SVM theory provides estimates of the worst case bounds on the LOOCV error, such as the fraction of training examples which become support vectors. However, since such bounds are necessarily loose, it can be useful for accurate model selection to compute the actual LOOCV error, if it can be obtained efficiently.

Our alpha seeding approach to LOOCV is as follows. First, train the SVM for all N examples. Denote the resulting alphas as β. For each of the examples (i) out of the full N, pretend in turn that that i-th one is not in the data set. [3] If β_i is already 0, then simply classify this i-th example as the full SVM does (and record if it disagrees with L_i). Otherwise, initialize the N alphas (α) to be those of β and set α_i to 0 (i.e. forget it). In that case, the equality constraint $\sum_{i=1}^{N} \alpha_i L_i = 0$ is violated, by a residual of magnitude β_i. To re-establish the equality, we must distribute that residual to some of the other alphas. Finally, after training the i-th SVM from the so-adjusted alphas α, we classify the i-th example (and record if it disagrees with L_i).

We have found that a simple and yet rather effective method is to redistribute the residual among all the *in-bound* alphas (i.e. those greater than 0 and less than C). A key motivation is that modern SVM trainers tend to work on in-bound alphas before rexamining at-bound ones. This is because generally once an alpha reaches 0 or C it will tend to stay there during the remainder of a SVM training.

We have explored various schemes for redistributing the residual among the in-bound alphas. One which routinely performs well, although not the best in every case, is to uniformly add an equal portion of β_i to each in-bound alpha α_j for which its corresponding example j is in same class as the hold-out (i.e. same label L_i). That is, add $\frac{\beta_i}{z}$ to each, where z is the number of other examples of that class with in-bound alphas. The exception is that if this causes some alpha to reach (i.e. want to exceed) the limit C, then any remaining residual is (uniformly) redistributed among the remaining in-bound alphas of that class, until all residual is gone. We call this scheme *uniform same-class residual redistribution*, and report results with it in Section 4.1.

3.2 GrowC: Quick Training for Large C

A more complex alpha seeding method involves training SVM's using successively larger C values. It is commonly observed in SVM literature that larger C values tend to require substantially more training time than smaller values. However, we theorized that initial training with a smaller C could quickly identify approximate alpha weights which later trainings with larger C's would be able to refine.

More precisely, let $S = [C_1, \ldots, C_n]$ where $C_i < C_{i+1}$ be a training schedule that produces correct alpha weights for C_n, the target value of C. We will refer to the training phase that uses C_i as S_i. The GrowC approach takes the alphas produced at the end of S_i and uses them as seeds for phase S_{i+1}.

The heart of any such strategy relies on determining an effective schedule for growing C. Our goal in this work is to establish that good schedules do exist, and we defer an in-depth investigation into automatically producing them to future work. The higher-order optimization method described in Section 3.6 for adjusting alpha values automatically as training progresses could also be used to choose appropriate intermediate C values.

Another key issue involves adjustments to the alphas between training phases. When moving from S_i to S_{i+1}, the range of allowable alpha values expands from $[0 \ldots C_i]$ to $[0 \ldots C_{i+1}]$. There are several options available. The alphas from S_i can be passed unchanged to S_{i+1}. Alternatively, the S_i alphas that are at C_i can be moved to C_{i+1}. A third alternative is to scale all of the alphas into the new range. Lastly, a more complex (possibly adaptive) method can be used to adjust only those alphas that are likely to move from their S_i values (as in Section 3.6). In Section 4.2, we compare the results of the first three options empirically and demonstrate the importance of good choices for alpha adjustment between training phases.

[3] For efficiency, we do not actually destroy the original data set, but instead have refined our SVM algorithms to allow ignoring one selected example during the QP optimization.

3.3 Kernel parameter via LOOCV

Another natural use of alpha seeding is for sequential SVM's over some range of settings for a kernel parameter. Previous work with Kernel Adatron SVM trainers [2] showed this to be effective, often not costing significantly more to train for a large number of parameter values than for the first one.

Based on our experience with this case, its effectiveness seems derive in part from the fact that the kernel values often do not change substantially under smaller parameter changes.

3.4 Adding new examples or features

For completeness, we mention that another promising use of alpha seeding might be for incremental online SVM's, in which the training set is extended with additional examples and/or input features. For example, one might imagine a forward feature subset selection approach in which at each round the candidate feature which most radically change the alphas so far in some fixed time limit is selected. However, we have no specific empirical results is this area yet.

3.5 Heuristically guessing initial alphas

Alpha seeds need not be based on previous trainings of very similar SVM's. For example, they could be based on geometrical arguments for why a given example is likely to be support vector or not, or likely to be at C (i.e. a noisy example). Guessing which examples will be at 0 or C can be particularly useful for many SVM training methods, since such at-bounds cases can often be ignoring in many iterations of those algorithms.

A particular method in this area which we have explored is training a SVM using a linear kernel and then using those alphas to seed training a SVM for some target nonlinear kernel. The intuition is that for problems which are only slightly nonlinear, such seeds can be very close to optimal for the nonlinear case as well. This idea is especially appealing given the substantial time savings possible for linear kernels, due to the feasibility of folding all N alphas into only D weights necessary to evalute the SVM output for any example in the linear case.

A further idea along these lines would be to do standard linear regression (or, Fisher discriminate analysis, in the case of classification per se), and then suitably convert the resulting D weights into N alphas seeds. Given that the complexity of these classic methods is $O(N \cdot D^3)$ whereas SVM training is roughly $O(N^2)$, this idea seems appealing. For example, one might use it to seed a linear SVM. However, the mapping from D to N is one-to-many and it is not yet clear whether there are any promising preferences on that space of mappings, other than the SVM bounds and equality constraints themselves (whose exact solution would require full SVM training, defeating the point of any heuristical seeding).

3.6 Higher-Order Optimization

The popular practical SVM algorithms are all gradient-based (e.g. SMO [8] and SVM^{light} [5]). Their popularity is in large part because an explicit N-by-N kernel Hessian could not fit in computer memory for large N greater than about 10,000.

However, based on our examinations of the behaviors of the alpha values during the course of many SVM gradient-based trainings, it appears that quite often some alphas change in steady monotonic ways for long sequences. For example, we noticed that sometimes within the first 25% of trainings of the MNIST [7] digit data, the relative ordering of the alpha values remained constant, with most having roughly constant slopes of change across the remaining 75% of the SMO training iterations. Those are exactly the sorts of cases that second-order information could optimize — allowing them to more directly jump to their final values.

These methods thus involve periodically checking the alpha values and noting how they are changing. This can be accomplished by training for successive 10-second (or so) intervals and examining the alphas after each interval.

We do not yet have strong empirical results in this area. Some alphas can indeed be helped to converge quicker using such a method, but for large-scale problems of interest the inefficient converge of the other alphas has tended to dominate the overall cost in experiments so far. Nevertheless, we mention this case

within the context of our overall framework because it seems to offer particularly interesting opportunities for meta-learning (i.e. using machine learning itself to learn how to better optimize).

3.7 Promoting Modularity

Before proceeding to our empirical results, we note that the previous case suggests the general utility of viewing the SVM training process in a more modular way than in current approaches. One could imagine any of the alpha seeding methods we have proposed being tightly integrated within any specific SVM training algorithm. The temptation to do so seems especially strong for more complex cases, such as those using second-order information. However, there can be great value in separating the overall SVM optimization task into alpha seeding and alpha optimization processes, even though where the line is draw can be somewhat arbitrary.

By maintaining such modularity, one can freely mix a variety of SVM training algorithms with a variety of seeding heuristics with greater ease. In other words, it can be useful to view some incremental changes to alphas values as being inspired by educated guesses (e.g. heuristics) and some by more logical inferences (e.g. gradients).

4 Examples

To empirically explore alpha seeding, we modified two common SVM algorithms, our implementation of SMO [8] (with improvements suggested by [6]) and the freely available SVM^{light} [5]. Our modifications enabled them to take seed alphas as arguments and begin training from that point on, instead of the default of zero alphas.

For our initial experiments to report in this paper we selected the UCI **Adult** data set, since a fair amount of related work with this data set has already been published using the SMO and SVM^{light} algorithms. In particular, for direct comparison we used Platt's discretized versions, consisting of 123 binary input attributes and various subsets of the full set of 32562 [8].

All tests were run on an 450Mhz Sun Ultra 60, with 2 Gigabytes of RAM.

4.1 LOOCV Results

For LOOCV tests, we used the smallest subset Platt reported on in his work [8], which consists of 1604 examples.

Figure 1 shows the cummulative run times for standard SVM (zero alphas for each of the N LOOCV retrainings) and our uniform same-class residual redistribution LOOCV alpha seeding method (as described in Section 3.1). Our method performs nearly 5 times faster in this test.

The training time for full data set was 2.86 secs and resulted in 714 out of 1604 examples being support vectors. The LOOCV training for each of the 714 hold-outs which were support vectors each took roughly the amount of time as that for full training: mean 2.943 secs, standard deviation .2923 secs, maximum 4.51 secs, and minimum 2.24 secs. Using our alpha seeding, training times for the support vector hold-outs were faster: mean 0.6452 secs, standard deviation .2245, maximum 1.54 secs, and minimum 0.22secs.

Both methods, of course, computed the same LOOCV error rate (16.55%), since their only difference is in speed of convergence. It is interesting to confirm that this rate is substantially below the (well-known to be loose) LOOCV error estimate bounds (44.51%) that the standard ratio of support vectors divided by the number of examples would suggest.

Figure 2 helps illustrate why our method performed so much better than a standard SVM non-seeded method. It plots all N training times (sorted from smallest to largest for each method). 714 of the examples required no training (because they were non-support vectors), indicated by many zero training times. Another 301 examples are treated as non-support vectors for the sake of this figure (i.e. assigned zero training time), because they had very small alpha values (near zero already). For the remaining 589 examples, there is substantially more area under the curve for the zero seeds than for the redistribution-based seeds. One can see that this is due to almost all zero-seed trainings requiring roughly same amount of time. Whereas using our alpha seeding method, a substantial number of the trainings involved times much smaller than

the mean. For all times, our seeded trainings were sigificantly faster than the full N example initial training (which took 2.86 secs).

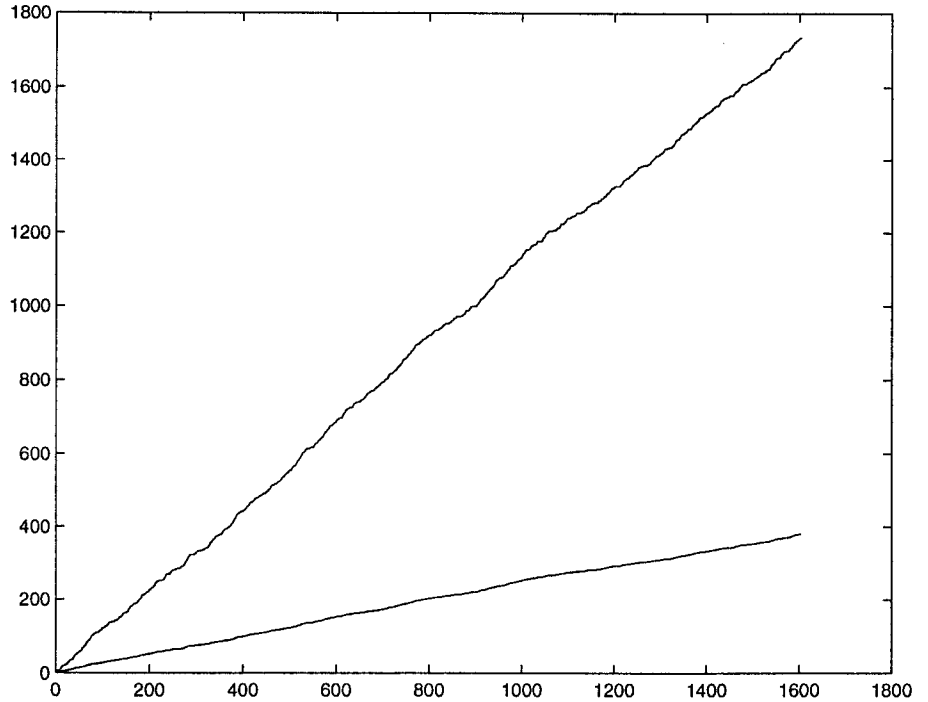

Figure 1: SMO cummulative training times for LOOCV on Adult1 data
Plots time (y-axis) for each of the $N = 1604$ LOOCV trainings (x-axis) after the i-th example is removed. Higher curve is for the standard SVM (start with alphas all zero). Lower curve is from using our *uniform same-class residual redistribution* LOOCV alpha seeding method, as described in Section 3.1. The total train times are 1733 secs and 380 secs (i.e. our seeding is 4.7 faster than zero seeding). For linear kernel, with C=1, for UCI Adult1 data set.

4.2 GrowC Results

For both our modified SMO and SVM^{light} algorithms, we experimented with several schedules for gradually growing C. In general, we observed that alpha seeding obtained dramatic reductions in total runtime for both algorithms. The particular Adult data set we used for these experiments is referred to as "Adult small" in [8], consisting of 11221 training examples.

We have verified that the number of bound and in-bound alphas we obtain correspond to those reported by Platt. All runs used a linear kernel and runtimes are averaged over five trials. We also made use of the cache that stores kernel computations, so that they need not be recomputed. This cache persists over each training phase S_i (after the first in a sequence of trainings), to make it comparable to training from scratch (where the cache is available throughout the course of training).

In Section 3.2, we outlined four options for how to seed S_{i+1} using the results of S_i. We here report on how the first three perform.

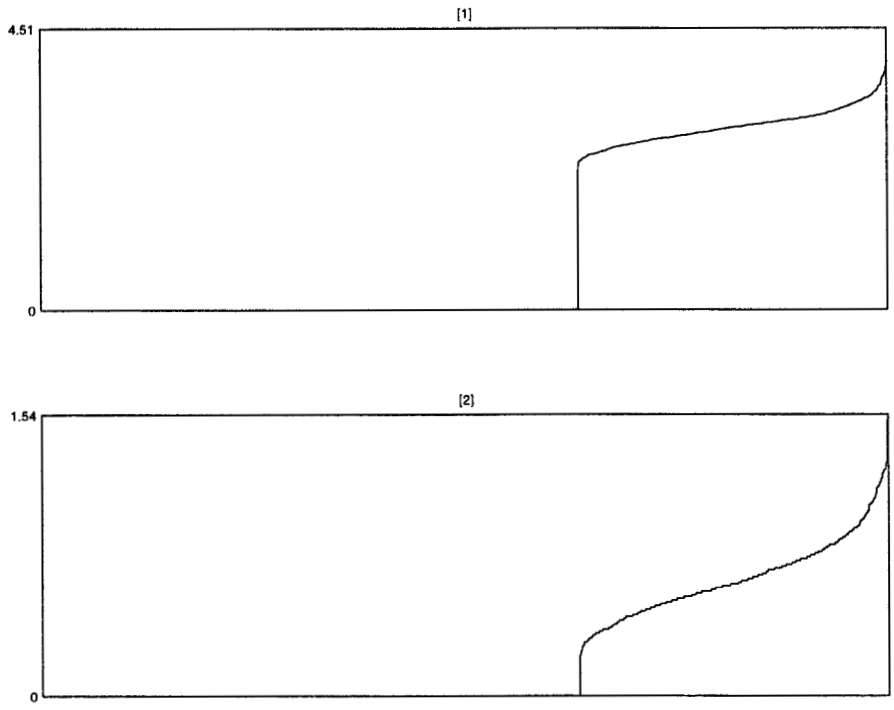

Figure 2: Sorted SMO training times for LOOCV on Adult 1 data
Top plot is for standard SVM (zero alphas), bottom plot is for our redistribution-based seeding method.

4.3 Direct Alpha Reuse

Using successively larger values of C and seeding each iteration with the alphas found at the end of the previous one does not always yield runtime benefits, as shown in Figures 3 and 4. For C values less than 0.3 for SMO and for *all* tested C values for SVM^{light}, it is actually more expensive to use this form of alpha seeding than to proceed from scratch. A smaller C_i restricts what possible alpha values can be explored, thus limiting the initial runtime, but when these alphas are used as seeds for S_{i+1} with a larger C_{i+1}, a lot of time can be spent adjusting them gradually into the larger range. This is especially true for alpha weights that are at C_i at the end of S_i – it is likely that they will end up being at C_{i+1} at the end of S_{i+1}, but it may take a long time to push them that far.

4.4 Scaling Bound Alphas

This observation leads naturally to the second option: at the end of S_i, change all alphas that have a value of C_i (the "bound" alphas) to the new C_{i+1} directly. The fact that an alpha is bound in S_i often indicates that it will be bound in S_{i+1}. If so, a lot of time can be saved by immediately jumping to the new boundary value, C_{i+1}. Figure 3 shows that this improves runtime for SMO over Direct Alpha Reuse, but can still (for C less than 0.1) be more expensive than training from scratch. Similar trends appear for SVM^{light} in Figure 4.

4.5 Scaling All Alphas

Our next option is to scale each alpha value produced by S_i into the new range allowed in S_{i+1}. This is accomplished by multiplying each alpha value by $\frac{C_{i+1}}{C_i}$. This has the effect of sending all alphas at C_i to

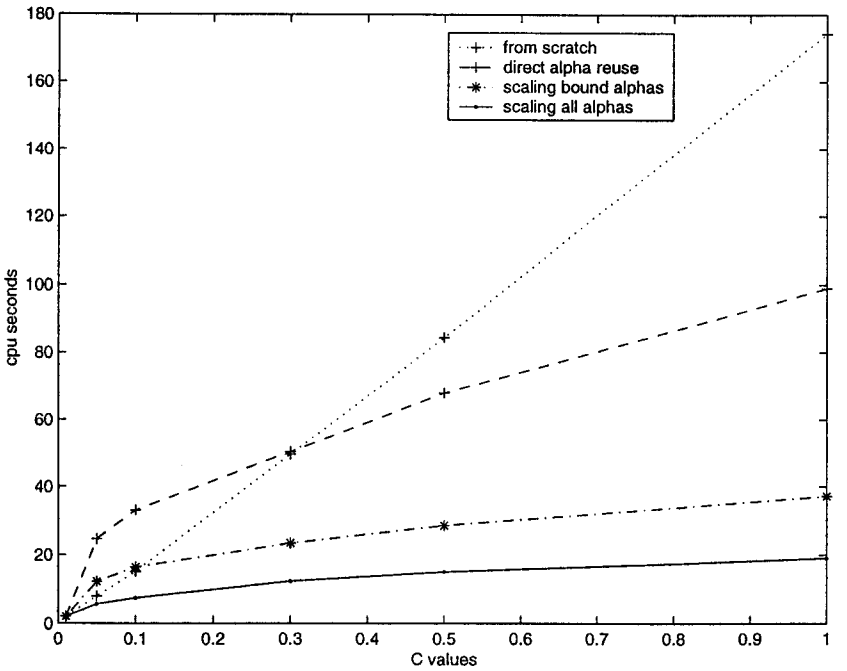

Figure 3: SMO runtime comparison on Adult data

the new value C_{i+1} and spreading the rest of the in-bound alphas into the new range. In addition, it keeps zero-valued alphas at 0. As shown in Figures 3 and 4, this strategy achieves the greatest improvements in runtime. Training SMO from scratch for $C = 1.0$ requires about 175 seconds. Scaling All Alphas requires just 19 seconds, a savings of 89% of the total runtime. For SVM^{light}, training from scratch requires 120 seconds, but Scaling All Alphas requires only 49 seconds (59% savings).

As noted above, the choice of schedule S impacts the effectiveness of alpha seeding. The seeding results in Figures 3 and 4 were all produced using schedule $S_1 = [0.01, 0.05, 0.1, 0.3, 0.5, 1.0]$, which was experimentally determined to work well with the Adult data. Figures 5 and 6 show the total runtime required when using various GrowC schedules, including:

$$\begin{aligned} S_1 &= [0.01, 0.05, 0.1, 0.3, 0.5, 1.0] \\ S_2 &= [0.01, 0.1, 0.5, 1.0] \\ S_3 &= [0.01, 0.1, 1.0] \end{aligned}$$

We here observe that more graduations in the schedule tend to yield greater overall benefits for SMO, but the reverse trend appears for SVM^{light}. Further investigation is required to fully understand what strategies for constructing training sequences are of most use to each algorithm.

Clearly, intelligent adjustments to alphas between training phases are essential. It is possible that better alpha adjustment strategies could result in even larger runtime improvements for alpha seeding. In addition, these results were all gained while using a linear kernel; other kernel types may require different alpha seeding (or C scheduling) strategies.

4.6 Larger C Values

Our results demonstrate significant improvements in performance for SMO for C values less than or equal to 1.0. Most of those C values are accompanied by a similiar improvement for SVM^{light}. However, it is not

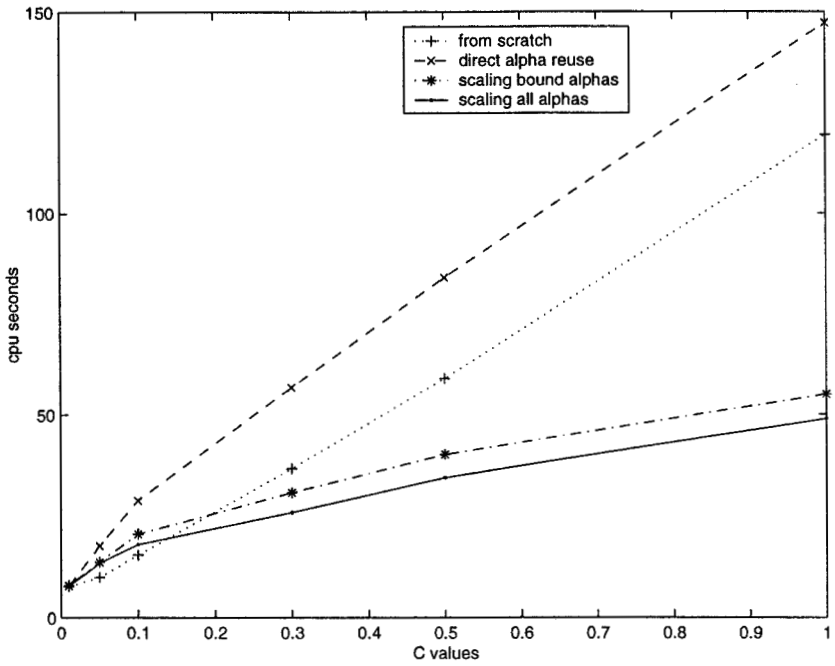

Figure 4: SVM^{light} runtime comparison on Adult data

usually possible to predict ahead of time what a good C value will be for a problem. Therefore, it is often observed that good performance over a variety of C values is important. In particular, large C values have been a challenge for SVM algorithms. In separate experiments, we were able to train on the Adult data with a C of 500 in under 85 seconds[4]. It took SVM^{light} and SMO over 10 minutes to train with such a large C.[5] Clearly, alpha seeding reduces these previously computationally-expensive trainings to reasonable durations.

The second benefit of using a seeding approach is that it can significantly reduce the time required to find a good value for C on a new data set. Instead of performing a series of trainings, all from scratch, with various values of C, it is instead possible to obtain results for *all* values of C by using a training sequence that contains each C value of interest. The alpha values are produced for each intermediate C_i and can be used to compute some test set accuracy obtained when using that value for C.

5 Conclusions

Our results suggest that alpha seeding is a feasible and promising way for speeding up SVM training. Although our speedups are often essentially constant ones, these factors are often much larger than the impact of other recently published methods for speeding up SVM's (e.g. bias intervals in [6] and "shrinking" in [5]). So they are of significant practical importance.

There are many directions for future work. One is to understand the nature of the best alpha seedings better, toward speedups that are typically more than nearly-constant ones (ideally, with amortized linear time cost for each SVM training). Another is to understand sensitivity issues, such as how close to the final alpha values do the seeds have to be, for significant speedup gains to be realized. Yet another is to develop means for automatically finding good growth schedules for any given task, for our GrowC method.

[4]The training sequence used was $[0.01, 0.05, 0.1, 0.3, 0.5, 1.0, 3.0, 5.0, 10, 15, 20, 30, 50, 100, 500]$.

[5]We terminated the training for each one at that point.

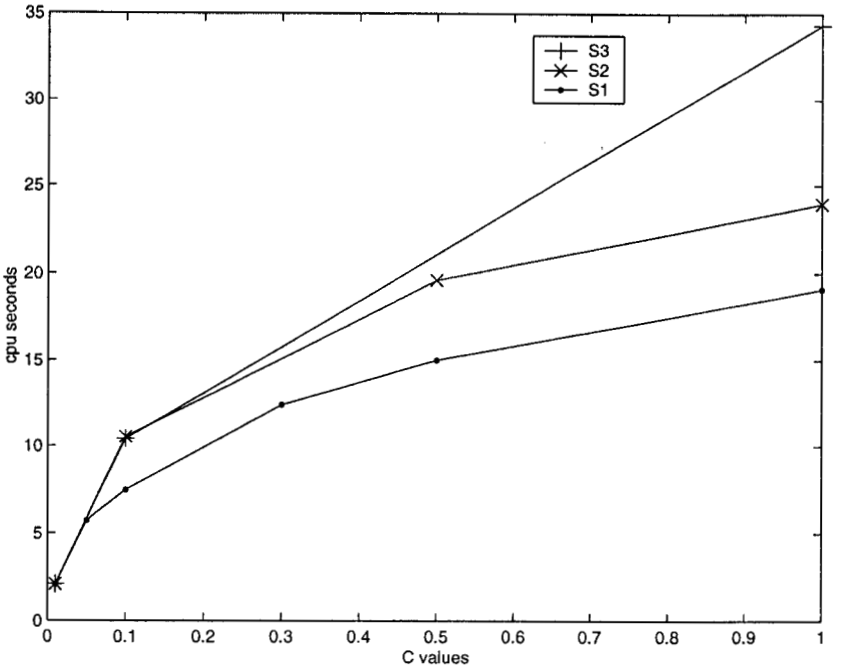

Figure 5: *SMO* runtime comparison on Adult data for different sequences

We also plan to contrast our efficient LOOCV alpha seeding approach over various C values against Leave-One-Out SVM's (LOOSVM's, [13]). Empirical results concerning the computational costs of LOOSVM's have not yet been published, so it is not clear which will be more effective – explicit search over specific C values as in our case versus folding the search for C within the optimization problem (as in LOOSVM's).

6 Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2), 1998.

[2] N. Cristianini, C. Campbell, and J. Shawe-Taylor. Dynamically adapting kernels in support vector machines. Technical Report NeuroCOLT Technical Report NC-TR-98-017, Royal Holloway College, University of London, May 1998.

[3] Dennis DeCoste and Michael Burl. Distortion-invariant recognition via jittered queries. In *Computer Vision and Pattern Recognition (CVPR-2000)*, June 2000.

[4] Isabelle Guyon. Online SVM application list. (See http://www.clopinet.com/isabelle/Projects/SVM/applist.html.).

[5] T. Joachims. Making large-scale support vector machine learning practical, 1999. In *Advances in Kernel Methods: Support Vector Machines* [10].

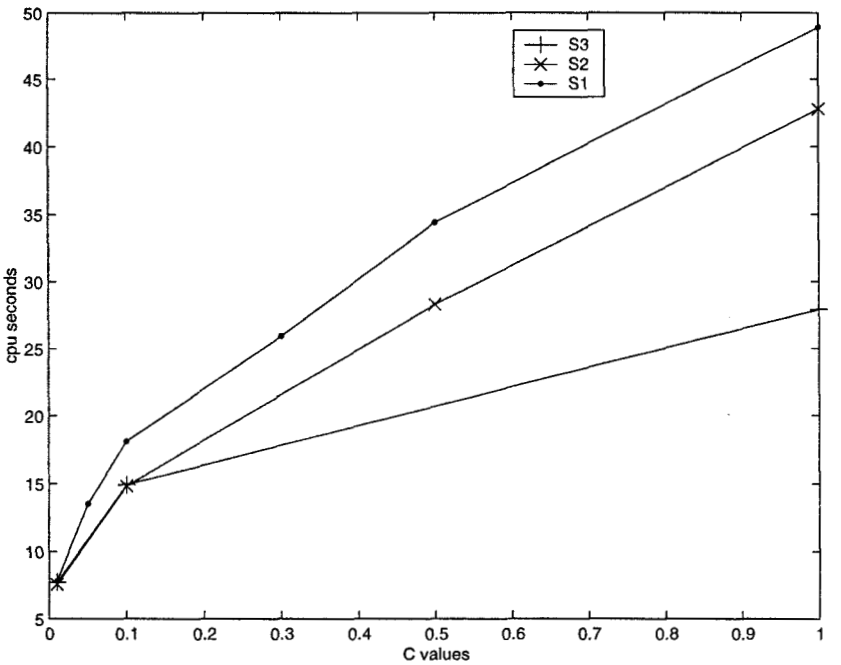

Figure 6: SVM^{light} runtime comparison on Adult data for different sequences

[6] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, and K.R.K. Murthy. Improvements to Platt's SMO algorithm for svm classifier design. Technical Report CD-99-14, Dept. of Mechanical and Production Engineering, National University of Singapore, 1999.

[7] Y. LeCun. MNIST dataset. (www.research.att.com/~yann/ocr/mnist/).

[8] John Platt. Fast training of support vector machines using sequential minimal optimization, 1999. In *Advances in Kernel Methods: Support Vector Machines* [10].

[9] B. Schölkopf, A. Smola, and K.R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. Technical report no. 44, Max-Planck-Institut for Biologische Kybernetik, Tübingen, Dec 1996.

[10] B. Schoelkopf, C. Burges, and A. Smola. *Advances in Kernel Methods: Support Vector Machines.* MIT Press, Cambridge, MA, 1999.

[11] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer Verlag, New York, 1995.

[12] K. Veropoulos, C. Campbell, and N. Cristianni. Controlling the sensitivity of support vector machines. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

[13] Jason Weston. Leave-on-out support vector machines. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.